Neural Style Transfer of Different Art Styles

Ayçe İdil Aytekin (21803718)
Cemal Gündüz (21703004)
Eren Şenoğlu (21702079)
Dora Tütüncü (21801687)

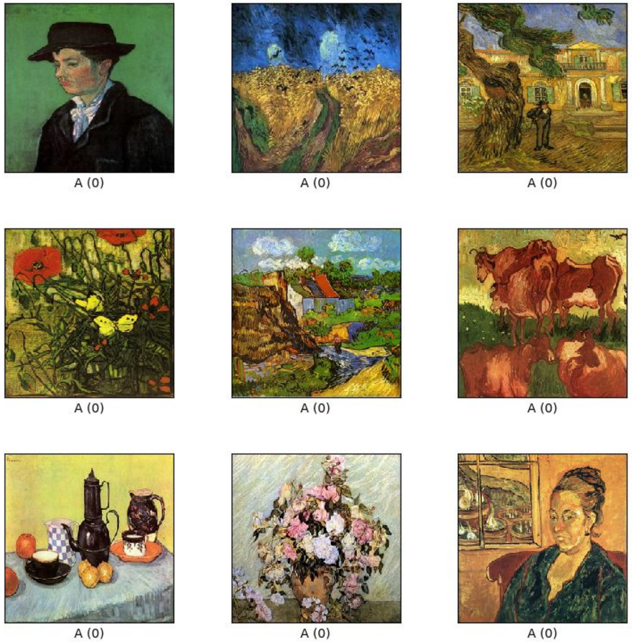
Introduction

- **Aim**: Improve the transformation quality with respect to **VGG-19**.
- Previously, VGG19 was used and Gatys’ paper was followed.
- Now, **CycleGAN** used for image-to-image translation. [1] is followed.
- Van Gogh’s paintings are used as style and landscape photos are used as content images.
- CycleGAN trains two generator models and two discriminator models.

The Dataset

“vangogh2photo” dataset is used. Dataset consists of 7.8k of samples. Samples are divided into four groups and 2 classes. First class Van Gogh’s paintings and the other class is real landscape photos. These classes divided into four groups as train and test.

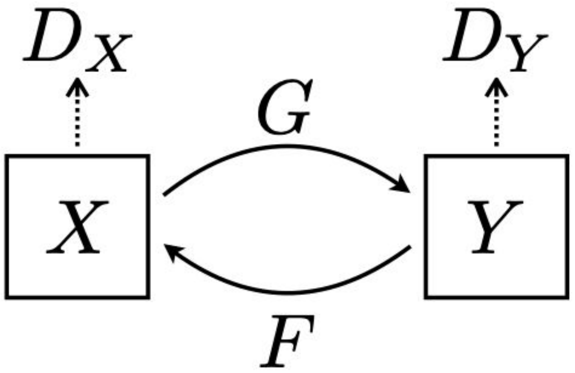
As preprocessing, images are normalized and augmented to possess correct shape for the algorithm.



CycleGAN

- learn a mapping G: X ⟶ Y
- learn a mapping F: Y ⟶ X
- Dx aims to distinguish between real images from domain X and F(Y), which are the fake X images
- Dy aims to distinguish between real images from domain Y and G(X), which are the fake Y images
- We aim to solve:

$$G^*, F^* = \arg\min_{G,F}\max_{D_x,D_y} \mathcal{L}(G, F, D_x, D_y)$$



Losses

- 3 types of losses:
  1. Adversarial Loss
  2. Cycle Consistency Loss
  3. Identity Loss

Adversarial Loss

- G tries to produce images G(X) similar to Y while Dy tries to distinguish real Y images from fake G(X).
- G tries to minimize this objective while Dy tries to maximize it
- Mean-square loss is used instead of negative log-likelihood as in the CycleGAN paper

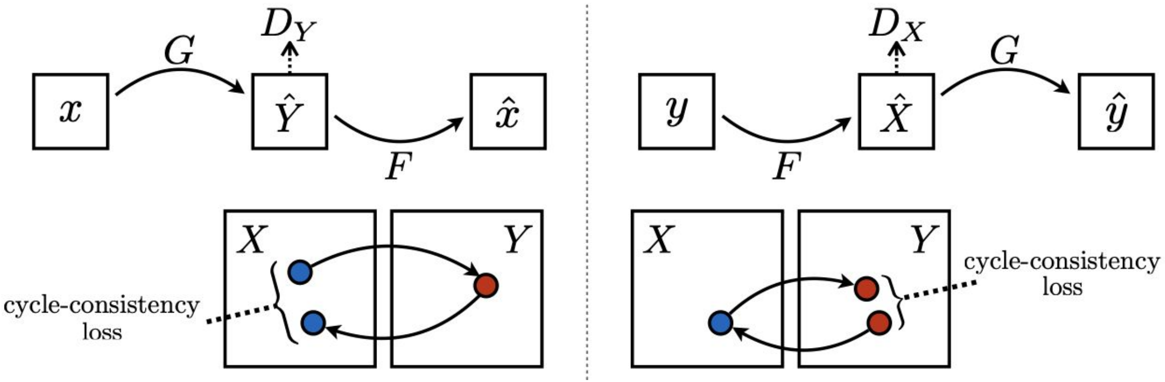
$$\min_{G} \max_{D_y} \mathcal{L}_{GAN}(G, D_y, X, Y)$$

$$\mathcal{L}_{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{data}(y)}[\log D_Y(y)] + \mathbb{E}_{y \sim p_{data}(x)}[\log(1 - D_Y(G(x))]$$
$$+ \mathbb{E}_{y \sim p_{data}(x)}[\log D_X(y)] + \mathbb{E}_{y \sim p_{data}(y)}[\log(1 - D_X(F(y))]$$

Cycle Consistency Loss

- The learned mappings also should be able to bring back the original image
- x ⟶ G(x) ⟶ F(G(x)) ⟶ ≈ x
- y ⟶ F(y) ⟶ G(F(y) ⟶ ≈ y
- L1 norm loss

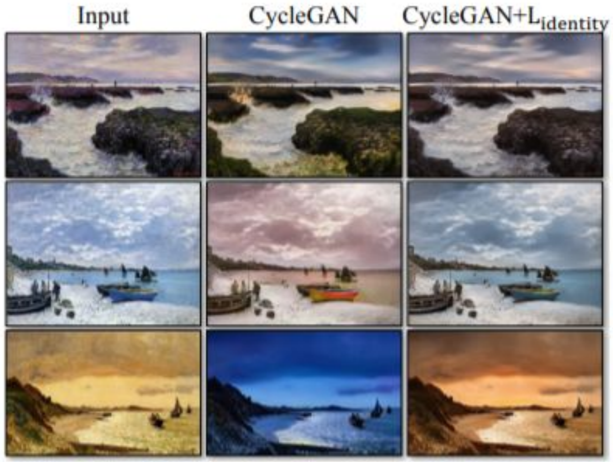
$$\mathcal{L}_{cyc}(G,F) = \mathbb{E}_{y\sim p_{data}(x)}\left[\left\|F(G(x)) - x\right\|_1\right] + \mathbb{E}_{y\sim p_{data}(y)}\left[\left\|G(F(y)) - y\right\|_1\right]$$



Identity Loss

- Identity loss is used to encourage the mapping to preserve color & composition btw input and output
- The method is to regularize the generator to be near an identity mapping when real samples of the target domain are provided as the input to the generator.
- If this Loss is not introduced, the tinting will arbitrarily change.

$$\mathcal{L}_{identity}(G,F) = \mathbb{E}_{y\sim p_{data}(x)}[\|F(x) - x\|_1] + \mathbb{E}_{y\sim p_{data}(y)}[\|G(y) - y\|_1]$$



Total Loss

Total loss equals to the sum of all distinct losses we decided to use which is:

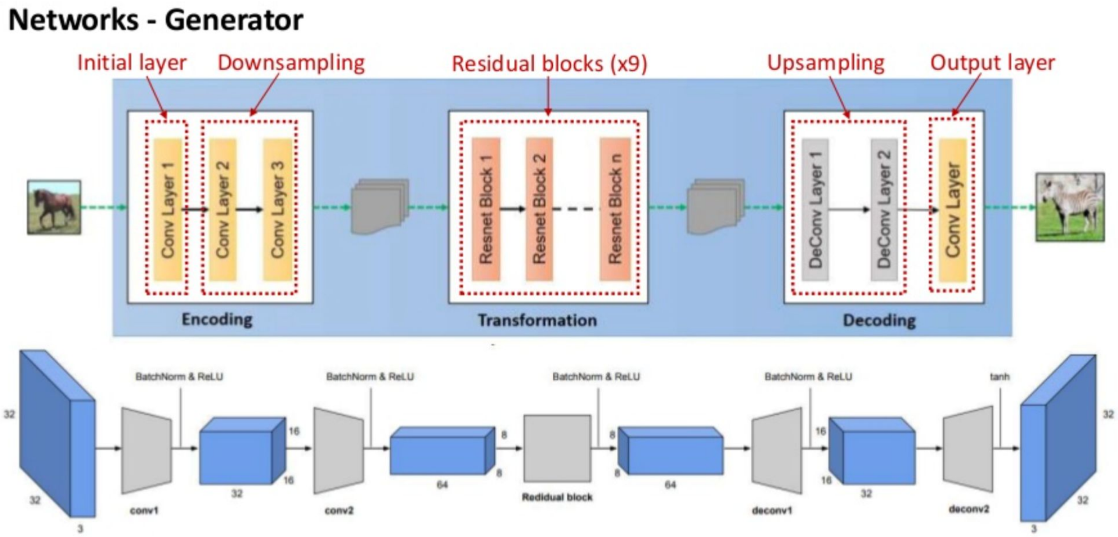
$$\mathcal{L}(G, F, D_x, D_y) = \mathcal{L}_{GAN}(G, D_Y, X, Y) + \mathcal{L}_{GAN}(F, D_X, Y, X)$$
$$+\lambda_1 \mathcal{L}_{cyc}(G, F) + \lambda_2 \mathcal{L}_{identity}(G, F)$$

Cycle loss weight $\lambda_1 = 10$

Identity loss weight $\lambda_2 = 5$

Generator

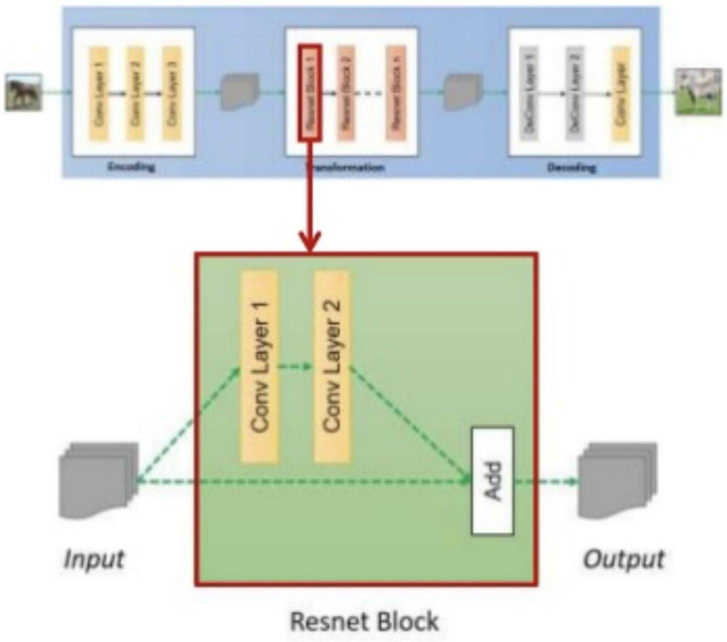
- Generators are used to transform class A images to class B images in cycleGAN.
- We used them to generate paintings and photos.
- Image is first downsampled with two convolution layers. Then processed through 9 residual blocks.
- Lastly, image is upsampled with two convolution layers.



Generator

```
class Generator(nn.Module):

    def __init__(self):
        super(Generator, self).__init__()

        # Initial convolution block
        model = [nn.Conv2d(3, 64, kernel_size=7, padding=3, padding_mode='reflect'),
                    nn.InstanceNorm2d(64),
                    nn.ReLU(inplace=True) ]

        # Encoding aka downsampling
        model = model + [nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
                        nn.InstanceNorm2d(128),
                        nn.ReLU(inplace=True),
                        nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1),
                        nn.InstanceNorm2d(256),
                        nn.ReLU(inplace=True) ]

        # Residual blocks
        for _ in range(9): #9 residual blocks
            model = model + [Residual(256)]

        # Decoding
        model = model + [ nn.ConvTranspose2d(256, 128, 3, stride=2, padding=1, output_padding=1),
                          nn.InstanceNorm2d(128),
                          nn.ReLU(inplace=True),
                          nn.ConvTranspose2d(128, 64, 3, stride=2, padding=1, output_padding=1),
                          nn.InstanceNorm2d(64),
                          nn.ReLU(inplace=True) ]
        # Output layer
        model = model + [nn.Conv2d(64, 3, kernel_size = 7, padding=3, padding_mode='reflect'), nn.Tanh()]

        self.model = nn.Sequential(*model)

    def forward(self, index):
        return self.model(index)

    def load(self, model):
        self.load_state_dict(torch.load(model, map_location=lambda storage, loc: storage)) #Load the model onto the CPU

    def save(self, model_path):
        torch.save(self.state_dict(), model_path)
```

Residual Block

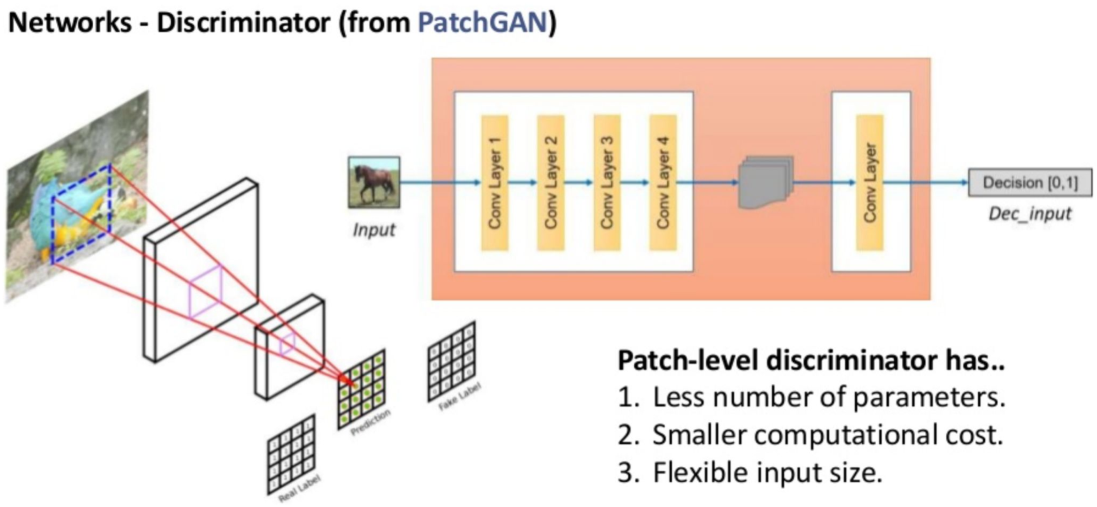


Training errors are expected to decrease as more layers added to the network. However, this is not the case for the real life practices and neural network reaches a point where training error starts increasing.

In order to fix this, residual blocks are introduced.

Discriminator

- 70x70 PatchGANs
- Discriminates the real and the fake overlapping patches
- Average pooling is applied at the end to acquire 1D labels
- Uses InstanceNormalization instead of BatchNormalization which standardizes each output feature map rather than across features in a batch
- Output is a map of probabilities showing each region of the image is original or fake



```python
class Discriminator(nn.Module):

    def __init__(self):
        super(Discriminator, self).__init__()

        model = [   nn.Conv2d(3, 64, 4, stride=2, padding=1),
                    nn.LeakyReLU(0.2, inplace=True),
                    nn.Conv2d(64, 128, 4, stride=2, padding=1),
                    nn.InstanceNorm2d(128),
                    nn.LeakyReLU(0.2, inplace=True),
                    nn.Conv2d(128, 256, 4, stride=2, padding=1),
                    nn.InstanceNorm2d(256),
                    nn.LeakyReLU(0.2, inplace=True),
                    nn.Conv2d(256, 512, 4, padding=1),
                    nn.InstanceNorm2d(512),
                    nn.LeakyReLU(0.2, inplace=True) ]
        model = model + [nn.Conv2d(512, 1, 4, padding=1)]
        self.model = nn.Sequential(*model)

    def forward(self, x):
       x =  self.model(x)
       return F.avg_pool2d(x, x.size()[2:]).view(x.size()[0], -1) #average pooling and squeezing

    def load(self, model):
        self.load_state_dict(torch.load(model, map_location=lambda storage, loc: storage))

    def save(self, model_path):
        torch.save(self.state_dict(), model_path)
```

Replay Buffer

- Used to train Discriminator
- If the buffer is not full, images are continued to be inserted.
- If the buffer is full:
  - Returns a previously stored image and puts the current image to the buffer with 50% probability
  - Returns the current image with 50% probability
- Reduces model oscillation

```python
class ReplayBuffer(): #used to train the discriminator
    def __init__(self, max_size=50):
        self.max_size = max_size
        self.data = []

    def push_and_pop(self, data): #adding or retrieving an image
        bulwark = []
        data = data.detach()
        for thing in data:
          if len(self.data) < self.max_size:
              self.data.append(thing)
              bulwark.append(thing)
          else:
              if random.uniform(0,1) > 0.5:
                  k = random.randint(0, self.max_size-1)
                  bulwark.append(self.data[k].clone())
                  self.data[k] = thing
              else:
                  bulwark.append(thing)
        return torch.stack(bulwark)
```

Learning Rate

- Learning rate is 0.0002 for 100 epochs
- Then linearly decays to 0

```
class LambdaLearningRate(): # Learning rate is flat till start_dec epochs, then it reduces to 0 linearly at end of the training.
    def __init__(self, total_ep, start_dec):
        self.total_ep = total_ep #200
        self.start_dec = start_dec #100

    def step(self, epoch):
        if epoch - self.start_dec <= 0:
          return 1.0
        else:
          return 1.0 - (epoch - self.start_dec)/(self.total_ep - self.start_dec)
```

```
optimizerGenerator = torch.optim.Adam(itertools.chain(genForward.parameters(), genBackward.parameters()), lr=0.0002, betas=(0.5, 0.999))
optimizerDiscriminator = torch.optim.Adam(itertools.chain(disPh.parameters(), disPa.parameters()), lr=0.0002, betas=(0.5, 0.999))
schedulerGen = torch.optim.lr_scheduler.LambdaLR(optimizerGenerator, lr_lambda=LambdaLearningRate(tot_ep, decay).step)
schedulerDisc = torch.optim.lr_scheduler.LambdaLR(optimizerDiscriminator, lr_lambda=LambdaLearningRate(tot_ep, decay).step)
```

VGG19 Results

Input Style Image

Input Content Image

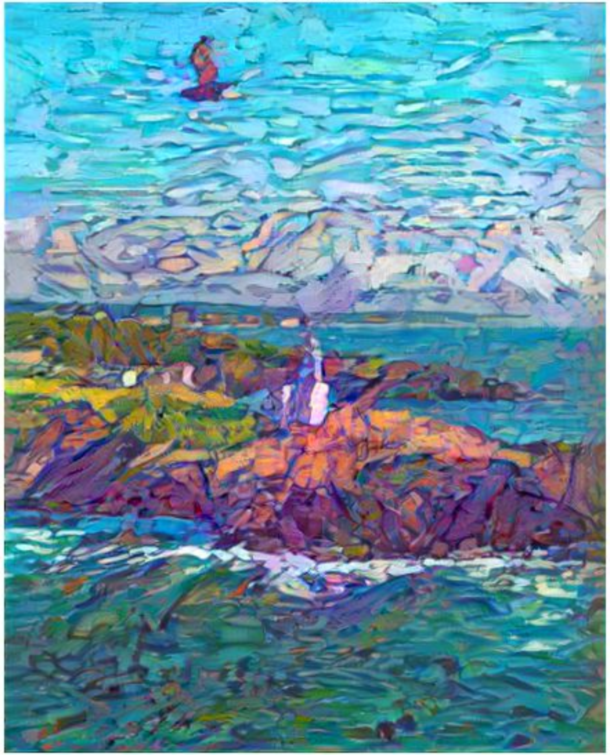
Stylized Content Image

VGG19 Results

Input Style Image

Input Content Image

Stylized Content Image



CycleGAN Results

original image

CycleGAN image

Stylized Content Image

style image

VGG19 image

original image

CycleGAN image

style image

Stylized Content Image

VGG19 image

Conclusion

VGG19 Style Transfer (Gatys followed)	CycleGAN Style Transfer
Transfers the style of a single selected piece of art	Transfers the style of the entire given collection, in this case Van Gogh
Pretrained model is used	Model is trained so more computational power is needed

References

[1] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks," *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017.